Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

312

J. van Leeuwen (Ed.)

Distributed Algorithms

2nd International Workshop
Amsterdam, The Netherlands, July 1987
Proceedings

Springer-Verlag

ON THE USE OF SYNCHRONIZERS FOR ASYNCHRONOUS COMMUNICATION NETWORKS

K.B. LAKSHMANAN
Department of Computer Science

AND

K. THULASIRAMAN
Department of Electrical and Computer Engineering
Concordia University
Montreal, Quebec H3G 1M8, Canada

ABSTRACT

A synchronizer is a mechanism that helps simulate a synchronous communication network on an asynchronous one. In this paper, we draw attention to certain difficulties that one may face in the implementation of a synchronizer mechanism, if the model of computation assumed does not permit delaying the processing of messages once received. To illustrate the issues, we present the design of a simple, time-efficient synchronizer. Using a synchronous breadth-first-search tree protocol, it will be shown that this synchronizer will not work correctly in all cases, because the simulation may not be exact. Similar difficulties arise in the implementation of other synchronizers proposed earlier in the literature, if the messages of the synchronous algorithm do not carry the pulse number. It is easy to suggest remedies for these difficulties which are at variance with the model of computation assumed. But, if we must stay within this model, we show that only the time-inefficient β-synchronizer known earlier can be corrected. We also present simple modifications to ensure this.

This research was supported in part by Concordia University under grant CASA-N67, the Natural Sciences and Engineering Research Council of Canada under grant A9194 and by the FCAR, Quebec under the Actions Spontanées grant 87AS2407.

I. INTRODUCTION

Consider a distributed computing system consisting of a number of processors interconnected through communication links. The processors do not share a common memory, have only local information and hence must communicate frequently to coordinate any computation to be accomplished. The interconnection network can be modeled by an undirected communication graph $G = (V,E)$, where nodes correspond to processors and edges to bidirectional communication links. Recently, a number of computational problems have been studied for such distributed systems, with potential applications in the design of data communication networks, distributed operating systems, distributed databases, etc. In such studies, two kinds of assumptions are typical, as far as the communication network is concerned. In an asynchronous network model [2-10,12,13], the exchange of messages between two neighboring processors is asynchronous in that the sender always hands over the message to the communication subsystem and proceeds with its own local task. The communication subsystem, we assume, will deliver the message at its destination, without loss or any alteration, after a finite but undetermined time lapse. On the other hand, in a synchronous network model [2,3,11,14], we assume the existence of a global clock, so that all network messages are sent only when the clock pulse is generated. Moreover, a message sent by any processor to its neighbor arrives at its destination before the next pulse. Understandably, these assumptions make the design, verification and analysis of protocols for synchronous networks much simpler, in contrast to those for asynchronous ones.

In the study of distributed algorithms, the complexity measures used to evaluate their performances relate only to the communication aspect, since messages are assumed to be processed in negligible computation time. The message complexity is the total number of messages transmitted during the execution of the algorithm. The time complexity is the time that elapses from the beginning to the termination of the algorithm, assuming that the delay in any link is exactly one unit of time. In an asynchronous network, a time complexity analysis provides a way of assessing the extent of parallelism present in the algorithm, and it should be noted that the assumption of unit delay in communication links is made only for the purpose of this analysis. The algorithm is expected to operate correctly under the actual assumption that the delay is finite but cannot be bounded.

A synchronizer is a mechanism that helps simulate a synchronous network on an asynchronous one. It is intended to be used as an additional layer of software, transparent to the user, on top of an asynchronous network, so that it can now execute synchronous protocols. Thus, with a synchronizer, the computation in an asynchronous network proceeds in rounds, trying to simulate the pulse-by-pulse activity of a synchronous protocol. For this purpose, a synchronizer basically generates a sequence of clock pulses at each node of the network, satisfying the following property: A new pulse is generated at a node only after it receives all the messages of the synchronous algorithm, sent to that node by its neighbors at the previous pulses [2]. Clearly, a synchronizer will require additional messages. Let C_{Pulse} and T_{pulse} denote the message and time requirements added by the synchronizer per pulse of the synchronous algorithm. Also, let C_{init} and T_{init} denote the message and time requirements of the synchronizer during its initialization phase. Then the complexities of the synchronous algorithm S and the asynchronous algorithm A, obtained by combining S with a synchronizer, are related as follows:

$$C_A = C_S + C_{pulse} * T_S + C_{init},$$

and

$$T_A = T_{pulse} * T_S + T_{init}.$$

Minimizing the message and time complexity overheads is the main issue in the design of synchronizers.

In [2,3], Awerbuch proposes the use of synchronizers as a general methodology for designing efficient distributed algorithms for asynchronous networks. In fact, he studies three different synchronizers - simply named α, β and γ synchronizers. The α-synchronizer is time-efficient and has $C_{pulse} = 0(m)$ and $T_{pulse} = 0(1)$, where m is the number of communication links or edges in the network. The β-synchronizer, on the other hand, is message-efficient and has $C_{pulse} = 0(n)$ and $T_{pulse} = 0(n)$, where n is the number of processors or nodes in the network. The γ-synchronizer combines the features of both α and β synchronizers and achieves $C_{pulse} = 0(nk)$ and $T_{pulse} = 0(\log_k n)$, where k is a design parameter such that $1 < k < n$. Thus, the γ-synchronizer exhibits a trade-off between its message and time complexities, which Awerbuch proves to be within a constant factor of the lower bound.

Even though it was not proposed as a general technique for designing asynchronous protocols, the use of synchronizer mechanisms for specific problems can be traced back to even earlier literature. In

[6], Chang uses a synchronizer mechanism, similar to the β-synchronizer – he calls his scheme a clocked network – for the problem of constructing a shortest path tree in an asynchronous network. Again, in [10], Gallager uses both α and β synchronizer mechanisms in the context of constructing breadth-first-search trees.

The issues discussed in this paper and the solutions we propose are intimately tied to the model of computation used. For asynchronous computations we follow the model used in [2-10,12,13] and for synchronous computations we follow the model used in [2,3,11,14]. These models are by far the most commonly used ones. In particular, we want to draw attention to assumption (d) in [13]. As stated there, in these models it is assumed that all messages received at a processor are stamped with the identity of the sender and transferred to a single common queue before being processed one by one. It is also assumed that the actions necessary for processing a message can all be performed in negligible computation time, without wait once started, and also, uninterrupted by the arrival of other messages. When processing of a message is complete, it is discarded. Thus, in this model of computation, it is not permitted to delay the processing of a message once received.

In this paper, we draw attention to certain difficulties one may face in the implementation of a synchronizer mechanism with the above restriction. To illustrate the issues, we present the design of a simple, time-efficient synchronizer in Section II. Using a synchronous breadth-first-search (BFS) protocol, it will be shown in Section III that this synchronizer will not work correctly in all cases, because the simulation may not be exact. However for some problems, it may be possible to modify the original algorithm slightly, and then employ this form of synchronizer. We also point out that similar difficulties arise in the implementation of the synchronizers proposed by Awerbuch, if the messages of the synchronous algorithm do not carry the pulse number. In Section IV, we present possible remedies for the difficulties faced in the implementation of the synchronizers and analyze their implications vis-a-vis the model assumed. A modification to the β-synchronizer is then proposed which leads to a correct synchronizer with $C_{pulse} = O(n)$ and $T_{pulse} = O(n)$. We also show that the α and γ synchronizers, however, cannot be corrected.

II. IMPLEMENTATION OF A SYNCHRONIZER

As mentioned earlier, in order to simulate the execution of a synchronous protocol, a synchronizer basically generates a sequence of clock pulses at each node of the asynchronous network. As stated by Awerbuch [2], the clock pulses generated must satisfy the following property: A new pulse is generated at a node only after it receives all the messages of the synchronous algorithm, sent to that node by its neighbors at the previous pulses. Awerbuch also asserts that this property ensures that the asynchronous network behaves as a synchronous one from the point of view of a particular execution of the synchronous algorithm. **Assuming this assertion to be correct**, a simple, time-efficient synchronizer can be designed as follows.

Every node sending a message to its neighbor ensures that all useful messages of the synchronous algorithm are sent before a GO message. Also, each node waits for an explicit GO message from each of its neighbors before it starts a new pulse. Thus, by the first-in, first-out property of the communication links, receipt of all GO messages signals that the node is ready to initiate actions corresponding to the next pulse. Observe that GO messages basically add O(m) message and O(1) time complexity overheads per pulse of the synchronous algorithm. In order to implement such a mechanism, we also need an initialization phase so that all nodes wakeup and send GO messages to their neighbors, thereby enabling every node to perform the first round of actions, corresponding to the first pulse of the synchronous algorithm. The propagation to all nodes of a WAKEUP message from the node initiating the algorithm can follow the protocol given by Segall [13]. Its message and time complexities are O(m) and O(n), respectively. Similarly, the fact that the computation is complete can be propagated to all nodes through a WINDUP message, with resulting message and time complexities of O(m) and O(n), respectively.

The synchronizer algorithm described above requires a data structure called "goreceived" to keep track at various stages of the set of neighbors from whom GO messages have been received. It is tempting to think that this set can be implemented at each node as a bit vector of size equal to the number of neighbors of that node. Unfortunately, this will not work. Consider a stage of execution when two neighboring nodes i and j have completed (k-1) pulses of the synchronous algorithm. Then, each one must have transmitted a GO message, permitting the other to go through the kth round. But, it is quite conceivable that before node i receives GO messages from all its neighbors and proceeds with the kth pulse, node j may receive all its GO messages and hence proceed to complete the actions corresponding to the kth pulse, thereby releasing another GO message to node i. Thus, some times, more than one GO messages could have arrived at node i from j, and hence the data structure "goreceived" should really be implemented as a multiset, permitting more than one copy of an element. It is clear, however, that node j cannot start the (k+1)th pulse, before node i completes its kth pulse and sends a GO message. Thus, at any point of time the number of pulses of activity gone through by two neighboring nodes can differ by at most one. This has two implications. The first is that at any point of time there could be at most two GO messages received at a node from any of its neighbors. The second implication is that over the entire network, the number of pulses of activity gone through by any two arbitrary nodes can differ by as much as the length of the shortest path between these two nodes in the undirected communication graph.

III. FAILURE OF THE SYNCHRONIZERS

Consider the execution of an arbitrary synchronous protocol combined with the synchronizer proposed in the previous section. We have already seen that if i and j are two neighboring nodes, there is the possibility that node j can complete the actions of its kth pulse after i completes (k-1) pulses but before it begins its kth pulse. This then implies a possibility of a message sent by node j in that pulse modifying some data structures maintained by node i, in the process altering what it would have done otherwise during the kth pulse. For example, some messages sent by the synchronous algorithm may not get sent in the simulated version. Since the set of messages transmitted by a node during a specific pulse of activity in a synchronous algorithm is only dependent on the values of all the variables maintained at that node just prior to the start of that pulse, one can freeze the computation of each node i just prior to the execution of its kth pulse and obtain its state Z_{ik}. The simulation of the synchronous algorithm with our synchronizer may not be exact in the sense that the sequence of states Z_{ik}, k = 1,2,... is not the same, at all nodes, for both cases.

As a specific example, consider the following synchronous protocol for constructing a breadth-first-search (BFS) tree of the communication graph [3,4,8,10]. In distributed computing, we assume that the algorithm will be initiated by a specific node s and that at the end of computation the BFS tree will be available in a distributed fashion, each node knowing its level number and its father, if any, in the tree.

The synchronous BFS protocol requires two kinds of messages - LABEL and ACK. LABEL messages carry a parameter and are sent to neighboring nodes to inform them of possible level numbers, whereas ACK messages are sent in response to LABEL messages, to inform that the corresponding LABEL messages have been handled suitably. The computation proceeds basically as follows. The source node labels itself at level 0 and sends down LABEL messages to each of its neighbors at the first clock pulse. A node when it receives the first LABEL message considers the sender of that message as its father, figures out its level number and sends down LABEL messages to each of its other neighboring nodes, if any. A node that receives a LABEL message and finds that it has no neighbor other than its father or that it has been labeled already with a level number, simply responds with an ACK message. Also, when a node receives ACK messages for each of the LABEL messages sent, it then sends an ACK message to the father node from which it received the first LABEL message. When the source node receives all its ACK messages, the computation is terminated. Recall that in a synchronous distributed protocol messages can be transmitted only when a clock pulse is generated. Thus, processing of messages does not trigger immediate dispatch of further messages. But suitable data structures are updated and information regarding messages to be sent at the next clock pulse are recorded.

Observe the simple structure of the synchronous BFS protocol. The first LABEL message to arrive at a node determines its level number, and there is no need to update this later. The correctness of the algorithm follows directly from the assumptions regarding the communication delays made under the synchronous network model. The fact that a message is transmitted only when a clock pulse starts, and that it reaches its destination before the next clock pulse starts ensures that a node with level number k will receive its first LABEL message only during the kth pulse. Clearly such a simple protocol is not possible for an asynchronous communication network.

Now, consider the execution of the synchronous BFS protocol presented above, combined with the synchronizer of the previous section on a 5-node asynchronous network shown in Fig. 1. Let s be the source

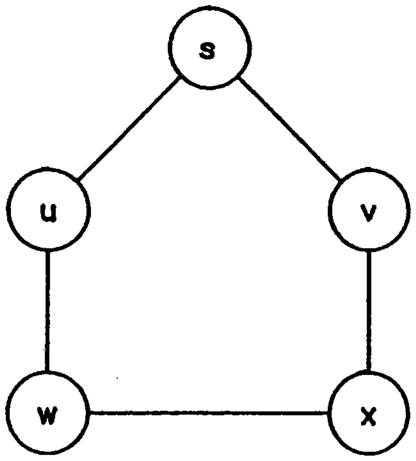

Fig. 1. Example network for BFS tree construction.

(root) node for the BFS tree to be constructed. Consider the stage of computation at which all five nodes have woken up and have transmitted GO messages to each of their neighbors. During the first pulse of activity, the source node s sends LABEL(1) messages to both u and v, followed by GO messages, permitting them to execute the second pulse. Nodes u, v, w and x, on the other hand, simply transmit GO messages to each of their neighbors. Suppose these messages are received properly, enabling both nodes u and v to be labeled at level 1. During the 2nd pulse of activity, node u will send a LABEL(2) message to w, followed by GO messages to s and w. Similarly, node v will send a LABEL(2) message to x, followed by GO messages to s and x. The other nodes, namely, s, w and x simply transmit GO messages to each of their neighbors. Now, suppose the messages from v to x are delayed arbitrarily. In the mean time, node w can receive the LABEL message from u, label itself at level 2, and, once the GO messages are received from both u and x, proceed with the execution of the 3rd pulse. During this pulse, it will transmit a LABEL(3) message to node x, which can cause node x to be labeled at level 3. Recall that this level number will not be revised even if the LABEL message from v is received subsequently. Clearly, the simulated version of the BFS protocol does not work correctly.

Observe that if the simulation is exact in the sense defined above, the correctness of a synchronous protocol guarantees the correctness of the simulated version. However, an exact simulation may not really be necessary for some algorithms to accomplish the required computation. Consider, for example, a modified synchronous BFS protocol where every LABEL message received is used to update the level number of the destination node, so that the effect of early arrival of LABEL messages with higher level numbers can be corrected. The algorithm will then become very similar to a synchronous version of the asynchronous protocol presented by Chandy and Misra [5] for the computation of a shortest path tree. It is easy to see that such a BFS synchronous protocol will work correctly when combined with the synchronizer of the previous section, even if the simulation is not exact. A formal proof of correctness can also be obtained along the same lines as in Chandy and Misra [5]. It must be emphasized here that as far as the execution of the modified BFS protocol on a synchronous network is concerned, there will never be a need to update the level number at any node. It is only when the protocol is combined with a synchronizer and used on an asynchronous network the modifications play a part in ensuring that a correct BFS tree is constructed. Such modifications may not be possible for arbitrary synchronous protocols. In any case, such a synchronizer cannot be viewed as a transparent layer of software placed on top of an asynchronous network to achieve a synchronous behavior.

The reason for the failure of the synchronizer proposed by us in the previous section can be attributed to the fact that the messages sent by a node j during its kth pulse of activity may arrive at a neighboring node i even before it starts its kth pulse. It is easy to show that the same situation can arise in the case of the synchronizers proposed by Awerbuch [2] also, unless corresponding pulse numbers are transmitted along with the messages of the synchronous algorithm and used as a basis for delaying the processing of some of the messages, as explained in Section IV.

A. α-synchronizer

To implement an α-synchronizer, Awerbuch requires each node to send out an explicit acknowledgment for each message of the synchronous algorithm it receives. Note that these acknowledgment messages do not increase the asymptotic message complexity of the synchronous algorithm. When a node receives acknowledgments for each of the messages it has sent out, it considers itself safe and sends out a SAFE message to each of its neighbors. A new pulse is generated at a node once SAFE messages are received from all its neighbors. It is easy to see that these SAFE messages play a role similar to that of the GO messages of our synchronizer. Thus, even with an α-synchronizer, messages corresponding to the kth pulse may be received at node i from a neighboring node j, even before it starts its kth pulse.

B. β-synchronizer

In contrast to an α-synchronizer where each node decides on its own to go ahead with the next pulse of activity once it has received SAFE messages from each of its neighbors, in a β-synchronizer there is a designated node which coordinates the computation. Initially, a leader is chosen and a rooted spanning tree is constructed. The synchronizer mechanism works as follows. After execution of a certain pulse, each node waits until it recognizes itself as safe as in the α-synchronizer. But, when a node learns that it is safe and all its descendants in the tree are also safe, it reports this fact only to its father. When the leader, being the root of the tree, recognizes that all nodes are safe with respect to a certain pulse, it notifies each node that they may all start a new pulse, again, via the spanning tree. Since all synchronizer related messages flow only along the spanning tree edges, the message and time complexities are both only $0(n)$ per pulse of the synchronous algorithm. Also, in this synchronizer, when any node starts its kth pulse of activity, it is guaranteed that all nodes in the network have completed $(k-1)$ pulses and that no messages of the synchronous algorithm corresponding to these $(k-1)$ pulses are still in transit. This is not so in the case of an α-synchronizer. But, if the β-synchronizer will have to be combined with an arbitrary synchronous protocol, the basic problem still remains. It is possible for node j to receive the notification from the leader, via the spanning tree, earlier than a neighboring node i, enabling node j to complete the actions of a particular pulse, before node i starts the corresponding pulse.

C. γ-synchronizer

To implement this synchronizer the network is first partitioned into clusters. A spanning tree is used for the coordination among nodes in a cluster. For communication between each two neighboring clusters, one communication link is chosen. In each phase of working of the γ-synchronizer, the β-synchronizer is applied separately in each cluster, and the cluster leaders communicate among themselves in a manner similar to that seen in an α-synchronizer. Thus, a γ-synchronizer is basically a combination of the α and β synchronizers, consequently suffering from the same problems mentioned above.

IV. POSSIBLE REMEDIES

For the correct working of an arbitrary synchronous protocol when combined with any synchronizer, it is necessary that before a node executes its kth pulse of activity, all messages sent to it by its neighbors during the (k-1)th pulse of activity must have been received and processed. All synchronizers proposed so far ensure this. It is also necessary that before a node executes its kth pulse of activity, no message sent to it by its neighbors during the kth pulse of activity should be processed, even if some have been received already.

There are several ways in which the requirements mentioned above can be met. Suppose, while processing a message it is valid to check up on some conditions and decide to handle the message right away or delay its processing by putting it back at the end of the queue. Then messages that arrive earlier than permitted can be effectively delayed. In the context of the synchronizer proposed in Section II, this can be achieved as follows. Whenever a message is received from j, a pending, unused GO message that has been received earlier from node j indicates that the processing of the current message has to be delayed. Similarly, if at each node, two queues are permitted for handling the incoming messages, the processing of some messages can be effectively delayed by queuing them up in the second queue. The processing of these messages can be taken up immediately after a new pulse has been generated at this node. Again, if there is one separate queue of incoming messages, for each of the communication links at a node, as assumed in [1], the requirements for the synchronizer can be easily met. The synchronizers proposed by Awerbuch [2] also work precisely this way by requiring each message to carry a pulse number, which can be used as a basis for deciding to delay the processing of messages that arrive earlier. Note that all these approaches are clearly at variance with the model of computation we have assumed for designing asynchronous distributed algorithms.

However, one valid approach will be to modify the β-synchronizer discussed in the previous section. Recall that in a β-synchronizer an elected leader coordinates the pulse-by-pulse activity of all nodes in the network. Here, when the leader decides to send down the PULSE message, notifying all nodes to start a new pulse of activity, it is guaranteed that all nodes have completed the previous pulse, no messages of the synchronous algorithm are still in transit, and that all nodes are ready to start a new pulse. But the problem is that the notification to start a new pulse of activity is propagated via a rooted spanning tree and sometimes this notification may arrive at a node later than a useful message of the synchronous algorithm transmitted by a neighboring node which has already completed its new pulse of activity. One way to remedy this will be to flood the network of this notification to start a new pulse, using the protocol of Segall [13], as is done for WAKEUP and WINDUP messages of our synchronizer. In other words, each node, when it first receives the PULSE message, informs each of its neighbors of that fact before proceeding with the execution of the new pulse. Subsequent PULSE messages received via other neighboring nodes can always be ignored. To accomplish this, a PULSE message must carry one bit of information that alternates between 1 and 0, indicating whether it corresponds to an odd numbered or an even numbered pulse. Correspondingly, each node must maintain a one-bit flag to keep track of the number of pulses of activity gone through. Based on this bit of information a node can decide to handle or ignore a PULSE message. As before, after completing a pulse of activity, a node may have to wait until it recognizes itself as safe before informing the leader so. This information can, of course, flow through the tree just as in a β-synchronizer. But observe that this solution then means that the message and time complexity overheads of the modified β-synchronizer are $C_{pulse} = O(m)$ and $T_{pulse} = O(n)$, respectively.

The message complexity overhead of the modified β-synchronizer can now be reduced, if the PULSE message is forwarded by each node only to those neighbors to whom it intends to send a useful message of the synchronous algorithm, during that pulse of activity. But, in order to be sure that each node receives the PULSE message at least once, it may have to be propagated via the rooted spanning tree also, as in a normal β-synchronizer. Now observe that the message complexity overhead resulting from the PULSE messages that flow along non-tree edges can be absorbed along with the message complexity of the synchronous algorithm, without affecting its asymptotic nature. Thus, this modified β-synchronizer which ensures the correct working of an arbitrary synchronous protocol also has only $C_{pulse} = O(n)$ and $T_{pulse} = O(n)$.

The question that remains is whether the α and γ synchronizers can also be corrected by finding simple modifications to them. Unfortunately, this is not so. First, observe that when a node u executes its kth pulse, it may send a message to a neighboring node v. Since node v is not permitted to delay the processing of this message, then, for simulation to be exact, it must also execute its kth pulse immediately, if it has not done so already. Therefore, before executing the kth pulse, node u must ensure that node v has completed its (k-1)th pulse and that no messages of the (k-1)th pulse are still in transit for itself or for node v. More importantly, if node v executes its kth pulse and sends a message to its neighboring node w, then w is also forced to execute its kth pulse immediately. As a result, before executing the kth pulse, node u must ensure that node w has also completed its (k-1)th pulse and that no messages of the (k-1)th pulse are still in transit for node w. Extending this argument, it is clear that for the correct working of an arbitrary synchronous protocol when combined with a synchronizer, it must be guaranteed that before any node executes its kth pulse every node in the network has completed its (k-1)th pulse and that no messages of the (k-1)th pulse are still in transit for any node. This argument proves that only a β-synchronizer can permit any arbitrary synchronous algorithm to be combined and run correctly on an asynchronous network.

V. CONCLUDING REMARKS

In this paper we have studied the concept of a synchronizer and identified some difficulties in implementing them. These difficulties are intimately tied to the fact that in the study of asynchronous distributed algorithms, the usual set of assumptions made do not permit delaying the processing of messages once received, while they require that messages be simply discarded after processing. However, a modification to the β-synchronizer, originally proposed by Awerbuch, is possible. This modified synchronizer which will work correctly in combination with any arbitrary asynchronous protocol has the same message and time complexity overheads per pulse as the original one. On the other hand, the time-efficient α-synchronizer cannot be corrected. It is clear from our discussions that the various synchronizers studied differ from each other in the level of synchronization each node demands before executing the next pulse. We have argued that the level of synchronization provided by the β-synchronizer is necessary for the simulation to be exact. However, such a high level of synchronization may not be necessary for certain classes of algorithms. In such cases, it may be possible to use other synchronizers as long as the corresponding synchronous protocols are defined carefully. For example, as we have pointed out earlier, the α-synchronizer and the one we have proposed using GO messages can both be used in conjunction with the synchronous version of the protocol given by Chandy and Misra [5] fo the shortest path problem.

REFERENCES

[1] H.H. Abu-Amara, "Fault-Tolerant Algorithms for Election in Complete Networks", Technical Report, Coordinated Science Laboratory, University of Illinois, USA, Feb. 1987.

[2] B. Awerbuch, "Complexity of Network Synchronization", J. Assoc. Comput. Mach., Vol. 32, No. 4, Oct. 1985, pp. 804-823.

[3] B. Awerbuch, "Reducing Complexities of the Distributed Max-Flow and Breadth-First-Search Algorithms by Means of Network Synchronization", Networks, Vol. 15, 1985, pp. 425-437.

[4] B. Awerbuch and R.G. Gallager, "A New Distributed Algorithm to Find Breadth-First-Search Trees", IEEE Trans. Info. Theory, Vol. IT-33, No. 3, May 1987, pp. 315-322.

[5] K.M. Chandy and J. Misra, "Distributed Computation on Graphs: Shortest Path Algorithms", Comm. Assoc. Comput. Mach., Vol. 25, No. 11, Nov. 1982, pp. 833-837.

[6] E.J.H. Chang, Decentralized Algorithms in Distributed Systems, Ph.D. Dissertation, (Also Technical Report CSRG-103), University of Toronto, Canada, 1979.

[7] E.J.H. Chang, "Echo Algorithms: Depth Parallel Operations on General Graphs", IEEE Trans. Software Engg., Vol. SE-8, No. 4, July 1982, pp. 391-401.

[8] G.N. Frederickson, "A Single-Source Shortest Path Algorithm for Planar Distributed Network", in Proc. STACS 85, Lecture Notes in Computer Science, Vol. 182, Springer-Verlag, Berlin, 1985, pp. 143-150.

[9] E. Gafni, "Perspective on Distributed Network Protocols: A Case for Building Blocks", in Proc. IEEE Military Communications Conference, Monterey, Oct. 1986.

[10] R.G. Gallager, "Distributed Minimum Hop Protocols", Technical Report LIDS-P-1175, Massachusetts Institute of Technology, USA, Jan. 1982.

[11] E. Korach, D. Rotem and N. Santoro, "Distributed Algorithms for Finding Centers and Medians in Networks", ACM Trans. Prog. Lang. Systems, Vol. 6, No. 3, July 1984, pp. 380-401.

[12] K.B. Lakshmanan, N. Meenakshi and K. Thulasiraman, "A Time-Optimal, Message-Efficient Distributed Algorithm for Depth-First-Search", Info. Proc. Letters, Vol. 25, No. 2, May 1987, pp. 103-109.

[13] A. Segall, "Distributed Network Protocols", IEEE Trans. Info. Theory, Vol. IT-29, No. 1, Jan. 1983, pp. 23-35.

[14] J.E. Van Leeuwen, N. Santoro, J. Urrutia and S. Zaks, "Guessing Games and Distributed Computations in Synchronous Networks", Technical Report SCS-TR-96, School of Computer Science, Carleton University, Canada, June 1986.